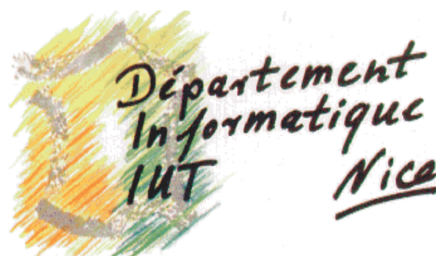
Inspecteur de code en JAVA

Tuteurs : Philippe Lahire et Pierre Crescenzo



Septembre 2002 – Décembre 2002
Licence Professionnelle des Métiers de l'Informatique

Sommaire

1.Introduction

2.Descriptiondusujet

2.1. Le contenu du projet.
2.2. Le langage JAVA.
2.3. Le projet JavInspector.

3.Réalisationduprojet

3.1. Les problèmes à étudier.
3.2. Les solutions apportées.
3.3. Le déroulement de la réalisation.

4.Etatd'avancementduprojet

4.1. Ce que fait le JavInspector.
4.2. Les classes et leur fonction.
4.3. La structure de données.
4.4. Les restrictions.

5.Résultatobtenu

5.1. Affichage des informations
5.2. Evolution possible.

6.Conclusion

1. Introduction

Afin de réaliser un projet de développement dans le cadre de la formation LPMI, nous avons pris contact avec M. LAHIRE et M. CRESCENZO pour qu'ils nous encadrent sur un sujet de leur choix.

Le sujet proposé, un inspecteur de code Java, dont une description plus précise est donnée en première partie de ce rapport, nous a permis de développer plusieurs compétences.

Tout d'abord il nous a fallu plonger dans l'apprentissage d'un nouveau langage, le JAVA, dont, malgré l'étendue de son utilisation de part le monde, nous n'avions que quelques notions de base. La première partie de ce rapport sera donc également consacrée à la description des outils dont nous avons fait usage et que nous maîtrisons un peu mieux aujourd'hui.

Puis nous verrons comment nous avons effectivement attaquer le problème, ce que nous avions à faire réellement, les choix de développement faits . C'est à dire tous les problèmes que nous avions soulevés au départ et les solutions que nous avons trouvées. Nous étudierons également la validité de ces solutions ainsi que leurs limites éventuelles. Et nous aurons quelques mots sur la façon dont s'est effectivement déroulée la réalisation du projet.

Nous montrerons par la suite l'état actuel du projet, les diverses fonctions que le JavInspector est en mesure de remplir et nous étudierons précisément son architecture, au niveau des classes, ainsi que son mode opératoire.

Enfin, il nous restera à conclure, pour faire un état des lieux à la fin de ce projet, voir ce qui a été fait, voir ce qu'il reste à faire et ce que nous avons tirés comme enseignements de ce projet.

2. Présentation

2.1. Le contenu du projet.

Le projet proposé consiste en la réalisation d'un inspecteur de code JAVA, lui-même réalisé en JAVA, capable de récupérer diverses informations sur le contenu d'une classe ou d'un ensemble de classes. Informations telles que les noms de classes, méthodes et attributs déclarés dans cette classe, par exemple.

N'ayant jamais encore développé sous JAVA, ce projet est donc pour nous à découper en deux étapes très différents :

- l'apprentissage du langage JAVA.
- La réalisation effective du projet.

2.2. Le langage JAVA.

2.2.1. Présentation

Cette étape était pour nous primordiale, car d'elle va découler notre compréhension précise du projet. En effet, notre projet étant très orienté sur le langage lui-même, plus qu'apprendre à simplement programmer avec ce langage, nous avions à bien nous imprégner de sa philosophie. Une bonne partie du temps de projet a donc du être consacrée à cette étude.

Nous verrons d'ailleurs par la suite comment notre manque d'expérience a entraîné certains retards sur le projet et certaines erreurs de réalisations.

2.2.2. JAVA en lui-même.

JAVA est un langage de programmation évolué, dont la syntaxe est proche du C++ et qui présente une série de caractéristiques remarquables :

- **JAVA est indépendant de toute plate-forme :**

Une applet programmée en JAVA fonctionnera sur la quasi-totalité des systèmes. Il n'est plus nécessaire de procéder à une compilation spécifique à la plate-forme comme c'était le cas avec C++.

- **JAVA est orienté objet :**

JAVA organise un programme en unités logiques constituées de données (variables et constantes) et de méthodes. Ces unités sont décrites sous forme de classes. Un modèle de ce genre permet au programmeur de créer des objets. Les objets disposent des variables et constantes de leur classe et peuvent être remplis avec des données qui leur sont propres. Les méthodes de la classe décrivent ce qui peut arriver aux données et régissent la communication avec les autres objets.

- **JAVA est simple :**

Les inventeurs du langage ont voulu supprimer tout ce qui s'avérait inutile. C'est ainsi qu'ont été abandonnés les pointeurs, le polymorphisme (il permet à une classe d'hériter de plusieurs autres en même temps), l'adressage mémoire direct des données, la surcharge des opérateurs, les structures (struct) et les unions (union).

- **JAVA organise la mémoire :**

Au moment de leur création, les nouveaux objets demandent qu'on leur réserve de la mémoire. Mais lorsque l'objet ne sert plus, il faut libérer la mémoire, ce qui est souvent délicat à gérer, et de ce fait, pas toujours effectué. JAVA dispose d'un excellent mécanisme : la récupération automatique de mémoire (garbage collector), qui restitue spontanément les zones de mémoire qui ne sont plus utilisées.

- **JAVA est sûr et rapide**

1) Le jdk

Pour écrire des programmes ou des applets en java, il faut disposer d'un système de développement : le Java Development Kit, communément appelé JDK, qui est le kit de développement basique que propose gratuitement la firme Sun Microsystem. A l'IUT, nous utilisons la version 1.4. Le Kit de développement comprend plusieurs outils, parmi lesquels:

- **javac**: le compilateur Java

  javac est un compilateur, c'est-à-dire qu'il transforme le code source en bytecode, un fichier binaire intermédiaire interprétable par la machine virtuelle sur n'importe quelle plate-forme.
  javac s'utilise avec la syntaxe suivante:

```
javac -g nom_du_fichier.java
```

L'option -g permet tout simplement d'inclure dans le pseudo-code des informations de débogage afin de pouvoir utiliser le débogueur jdb.

- **java**: un interpréteur d'applications (machine virtuelle)

  L'interpréteur java est une machine virtuelle fonctionnant en mode texte, c'est-à-dire sans interface graphique. Sa syntaxe est la suivante:

  ```
  java nom_du_fichier
  ```

- **applet viewer**: un interpréteur d'applets

  appletviewer a pour but de pouvoir visualiser l'exécution d'un applet (il est aussi possible de la visualiser sur un navigateur compatible Java, comme Internet Explorer 4 ou supérieur, Netscape Navigator 4 ou supérieur, HotJava...). Sa syntaxe est la suivante:

  ```
  appletviewer nom_de_l_applet.class
  ```

- **jdb**: un débogueur

- **javap**: un décompilateur, pour revenir du bytecode au codesource

- **javadoc**: un générateur de documentation

  JavaDoc est un utilitaire permettant de créer une documentation au format HTML à partir d'un ou plusieurs programmes, grâce aux commentaires prévus à cet effet (/**) incorporés dans le code.

- **jar**: un compresseur de classes Java

  Jar est un utilitaire permettant de compresser les classes Java afin de réduire leur taille et de rendre leur téléchargement plus rapide.

2.2.3. Principaux outils utilisés.

Les principales bibliothèques mises à notre disposition par JAVA dont nous avons eu à tirer partie, hormis les plus classiques et indispensables, furent les suivantes (il convient de noter que dans un souci de portabilité il nous était impossible de nous servir de bibliothèques propriétaires telles celles fournies par Microsoft.) :

- La bibliothèque SWING , bibliothèque graphique, et, plus particulièrement dans cette bibliothèque :
    - La classe Jtree : classe à partir de laquelle nous avons pu créer notre arbre de représentation des données.
    - La classe Jframe pour la création de fenêtres.
    - La classe JfileChooser pour la fenêtre de dialogue nous permettant la sélection des fichiers à inspecter.

- La classe Class : classe regroupant toutes les informations dont nous avions besoins sur les classes elle-mêmes, dans le même ordre d'idées nous avons utiliser les classes :
    - Field : informations relatives aux Champs.
    - Method : informations relatives aux méthodes.

- La class ClassLoader : qui est une classe permettant le chargement dynamique d'une classe éloignée ou hors du projet (voir description détaillée dans le chapitre 3.2.1.).

2.3. Le projet JavInspector

Comme décrit précédemment, le JavInspector est un inspecteur de code JAVA. Prévu au départ comme un analyseur syntaxique de code source, cette solution trop compliquée pour notre niveau de connaissance et surtout ne présentant peu d'intérêt au niveau programmation, a été abandonnée en cours de projet pour un analyseur de code compilé.

Pour cela, il sera utilisé l'ensemble des classes fournies par JAVA telles que nous les avons décrites précédemment, notamment celles qui permettent de récupérer toutes les informations désirées sur une classe chargée.

La technique utilisée pour charger cette classe et en récupérer les informations sera traitée au chapitre suivant.

3. Réalisation du projet.

3.1. Les problèmes à étudier.

Le premier problème consiste en la récupération du maximum d'informations concernant la classe étudiée. L'utilisation de la classe « *Class* » fournie par JAVA est d'une grande facilité mais possède ses limites :

- Elle ne récupère que les informations d'une classe chargée en mémoire, ce qui pose naturellement le problème du chargement dynamique de la classe que nous désirons étudier en mémoire pour pouvoir accéder à ses informations.

3.2. Les solutions apportées.

3.2.1. Chargement dynamique des classes étudiées.

La seule solution proposée par le langage JAVA consiste en la création d'un « Class loader » chargeant la classe passée en paramètre.

❖ Qu'est-ce qu'un Class loader ?

Le Class loader est un outil – en réalité une classe – qui s'occupe de rapatrier dans l'environnement d'exécution Java le code manquant et nécessaire. Une instance particulière s'occupe de charger dans la machine virtuelle les différentes classes au fur et à mesure des besoins.

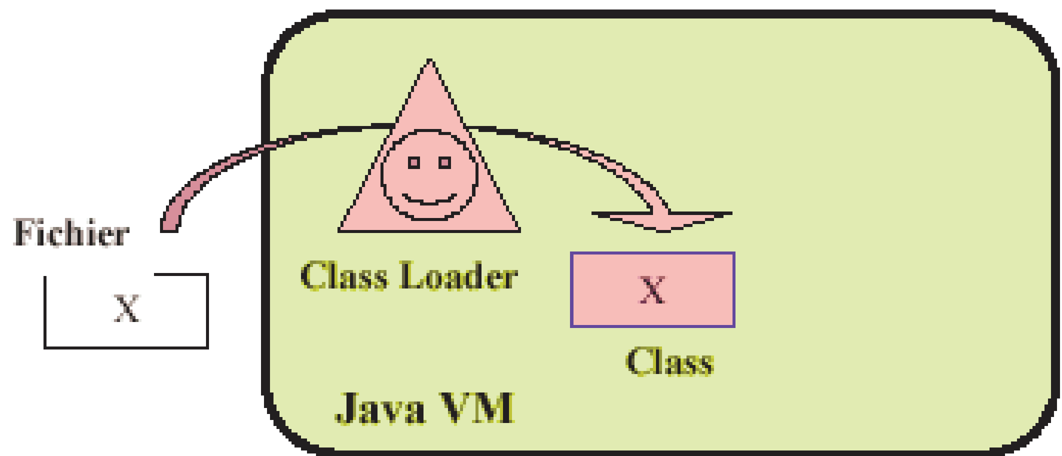


Figure 1 : Chargement en mémoire

Schéma : © Ph. PRADOS

Une classe particulière de notre application s'appliquera donc à créer un Class loader spécifique, différent de celui créé par défaut par l'exécution de notre logiciel, chargé de mettre en mémoire les classes inspectées.

3.3. Le *déroulement de la réalisation.*

3.3.1. Problèmes rencontrés et leur origine.

Au cours du déroulement du projet, plusieurs phases se sont succédées, des phases de progression rapide, de stagnation puis de recul dans l'avancement du projet. Il y a deux origines directes de ce déroulement un peu chaotique : le manque d'expérience dans l'utilisation de l'outil JAVA, et ce qui en découle un peu également, une interprétation parfois erronée des spécificités du projet.

3.3.1.1. Problèmes d'ordre technique.

La maîtrise d'un langage et de tous ses différents aspects nécessite bien évidemment plus de quelques semaines d'apprentissage autonome. Il nous est donc fréquemment arrivé d'utiliser des outils peu ou pas appropriés pour réaliser certaines fonctions du logiciel.

De ce fait, le développement a été fréquemment ralenti, voire arrêté pour repartir en arrière, voyant que les solutions trouvées ou les outils exploités ne nous permettaient pas de parvenir à nos fins.

3.3.1.2. Problèmes d'ordre interprétatif.

Du fait de ces lacunes techniques, nous nous sommes parfois égarés dans des tentatives de développements compliqués, n'ayant également pas bien cernés les attentes sur les fonctionnalités du logiciel.

Nous sommes conscients de l'écart existant entre la version finale du projet et la version attendue originellement.

Notamment en ce qui concerne l'utilisation d'un « Visiteur » réel. Cet aspect n'ayant été cerné que trop peu de temps avant la remise du présent rapport.

Nous expliquerons plus en détails dans la partie « Restrictions (4.3.) » ce qu'est un Visiteur et ce qu'aurait été son utilité.

4. **Déroulement de l'application**.

4.1. Ce que fait le JavInspector.

Le JavInspector va permettre à l'utilisateur de choisir une classe grâce à un explorateur de fichier. Il peut choisir une classe, un ensemble de classes ou un répertoire de classe.

Une fois ce choix effectué, le JavInspector charge dynamiquement les informations relatives à la classe sélectionnée dans une structure de donnée précise. Celle–ci est à son tour retranscrite sous une forme d'arbre où sont divisées méthodes et champs faisant partie de la classe. Une subdivision de ces informations, à savoir si ils sont hérités ou natifs, est également effectuée.

De plus, le JavInspector récupère les informations sur l'héritage de la classe sélectionnée et parcourt à l'envers la chaîne de tous les parents (jusqu'à Java.lang.object) et affiche cette chaîne sur l'arbre résultat. Chacune de ces classes ayant été au préalable transformée en objet classe, le même traitement leur est effectué : récupération des informations sur les méthodes et champs.

4.2. Les Classes et leur fonction

Voici la liste des classes ainsi que leurs fonctions dans le déroulement du logiciel, elles sont données dans l'ordre de création :

- Classe JavInspector : Classe principale du programme, elle s'occupe d'afficher la fenêtre de départ, crée le menu, les différents panneaux de réception d'informations et affiche un arbre vide.
  C'est par le menu « File » que l'on peut enclencher la recherche en ouvrant une boite de dialogue, instanciée au préalable par la classe « Fichier ».

- Classe Fichier : Classe événementielle déclenchée lors de l'utilisation du menu « File », permet donc la sélection des fichiers « *.class » à charger grâce à la ligne de Menu « Open File… ».
  Cette classe prend une information sur la fenêtre actuellement ouverte et le panneau où afficher le nouvel arbre à créer.
  Une fois celui-ci fait elle l'affiche et met à jour la fenêtre.

- Classe OuvrirBoite : Cette classe va commander l'ouverture d'une boite de dialogue pour la sélection des fichiers de type *JFileChooser*. Deux fonctions permettent de récupérer le répertoire de sélection et/ou les fichiers sélectionnés.

- Classe Visiteur : La classe qui va mettre en mémoire le fichier « .class » sélectionné pour que l'on puisse accéder à ses informations à l'aide des outils de la classe « Class ».
  Cette classe prend en paramètres les données issues de OuvrirBoite, c'est-à-dire le nom du ou des fichiers et le répertoire où ils se situent et s'occupe de les transférer à la classe « Chargeur ».

- Classe Chargeur : Implémentation d'un « ClassLoader », cette implémentation est nécessaire pour pouvoir permettre le chargement dynamique de la classe sélectionnée. Cette classe surcharge la méthode *FindClass ()* afin que nous puissions lui indiquer où chercher les classes désirées.
  Dans cette surcharge, on peut donc passer en paramètres les informations récupérées précédemment sur le lieu de stockage physique du fichier à charger. Il nous retourne une instance de type « Class » à partir duquel la classe « MaClasse » va pouvoir rechercher les informations dont elle a besoin.

- Classe MaClasse : créée pour contenir les informations générales relatives à la classe étudiée, elle contient également des « pointeurs » vers les listes de méthodes, de champs, ses ancêtres (déclarés comme des objets de types MaClasse également) ou les interfaces qu'elle implémente. C'est son instanciation qui enclenche le chargement complet de la structure de données.

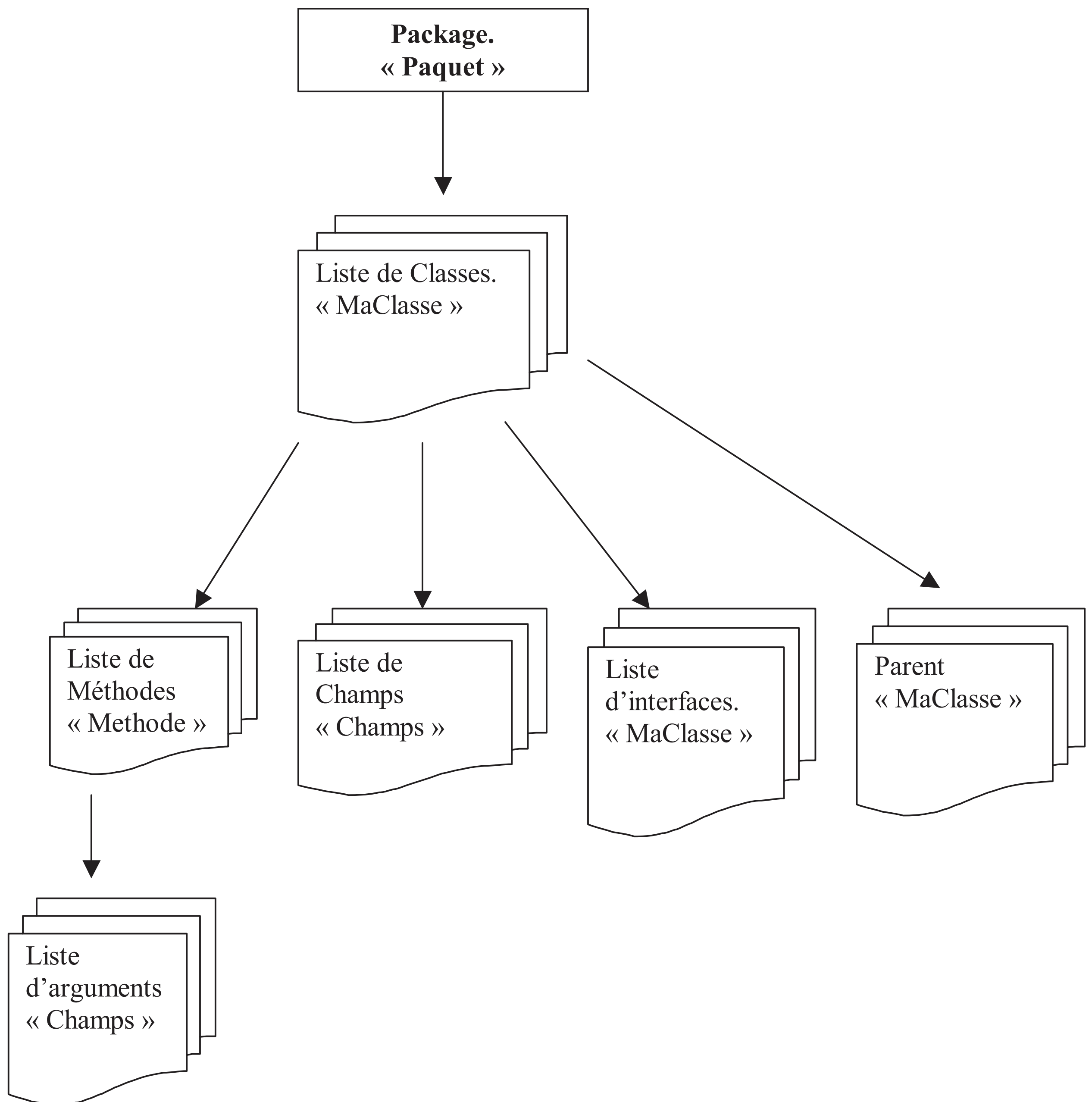
- Classe Methode : contient quand à elle les informations relatives à une méthode, ces informations sont principalement le type de retour, la liste d'arguments et son nom. Elle contient également un « flag » indiquant son appartenance directe à la classe ou son héritage.

- Classe Champs : très proche de la classe Methode, elle regroupe quant à elle tout ce que l'on a à savoir sur les champs de la classe précisée.

- Classe Paquet : regroupe les classes appartenant à un même package. Est nulle (« vide ») si la classe n'est pas dans un package.

- Classe Larbre : Cette classe s'occupe de générer l'arbre à partir des informations récupérées dans les classes précédentes. Il parcourt l'intégralité de la structure de données et crée à l'aide de la bibliothèque *Jtree* un arbre de visualisation de ces données.
  Il parcourt également l'arborescence supérieure de la classe constituée par ses « ancêtres ».

4.3. La structure de données



4.4. Restrictions

Nous entendons ici par restrictions les choix qui ont dû être faits dans ce développement à partir des propositions de départs.

4.4.1. Propositions de départ.

4.4.1.1. Le Visiteur

La notion de Visiteur repose en fait sur l'idée de l'utilisation d'une classe espion, placée directement (non dynamiquement) dans le répertoire à étudier et dont la fonction serait double :

- Récupérer des informations sur les fichiers présents dans le répertoire : à savoir la liste des fichiers .class qui s'y trouvent.

- Utiliser sa présence dans le même répertoire pour récupérer une partie des informations non accessibles : les champs ou méthodes définit comme « protected ».

4.4.1.2. Présentation graphique

La présentation, assez succincte, devait originellement présenter un deuxième écran d'affichage plus conséquent sur les classes visitées.

Il devait également figurer un panneau de paramétrage des types d'informations souhaitées et de la façon de les affichées dans l'arbre de visionnage.

4.4.2. Pourquoi ce retard ?

Suite à un problème d'interprétation du projet proposé, toute la première partie de notre travail sur le logiciel a du être recommencée.

En effet, nous pensions à l'origine devoir faire un analyseur syntaxique de code source JAVA, c'est-à-dire un véritable parseur capable de récupérer les informations nécessaires à partir d'un fichier non compilé.

Ce logiciel ne devant alors pas utiliser de « ClassLoader » ni les fonctions données par la classe « Class ».

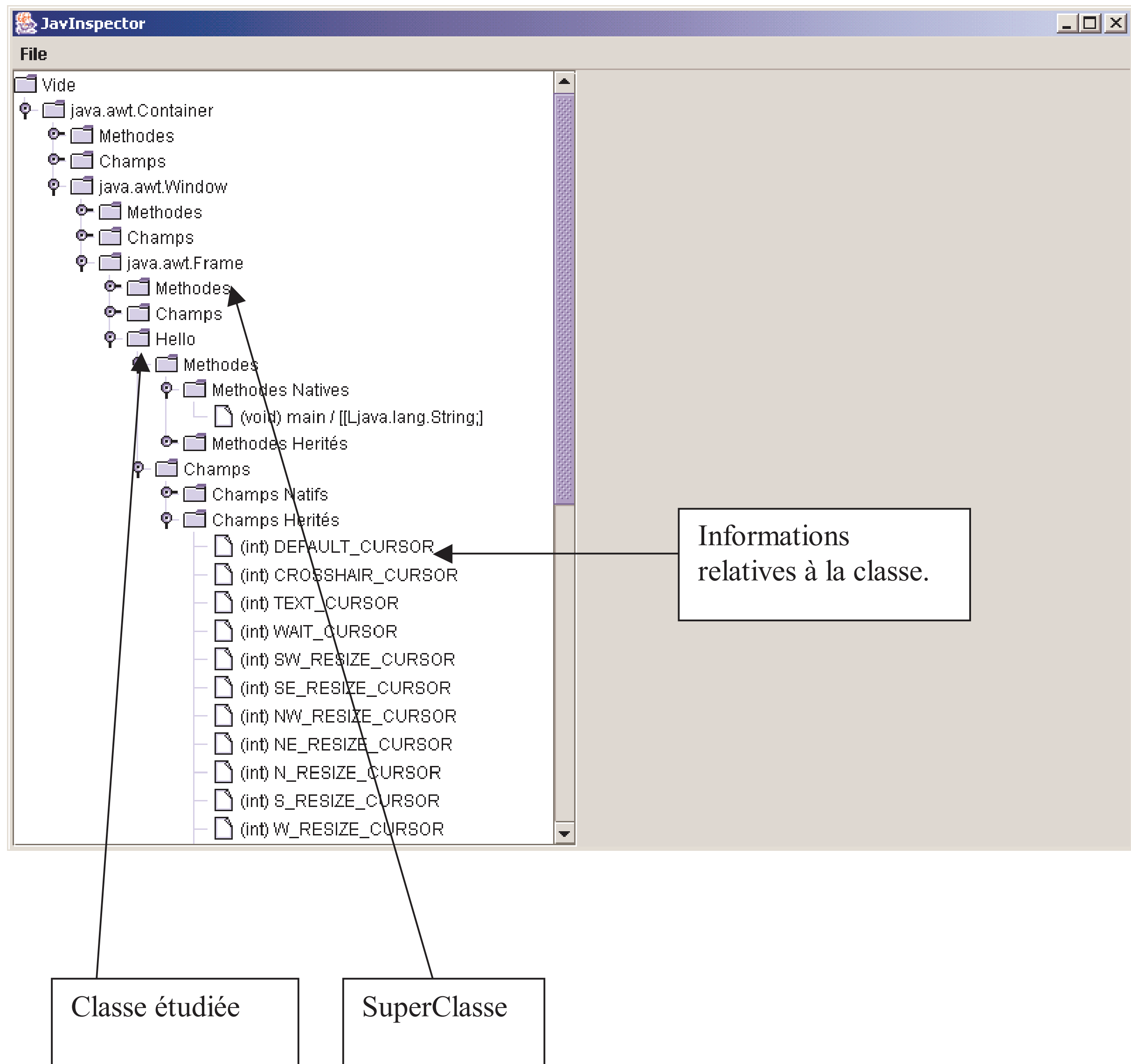
De plus, nous avions mal compris le concept de « Visiteur », fichier qui n'était pas à charger dynamiquement mais à placer dans le répertoire à

inspecter. Nous pensions justement devoir réaliser toute l'opération directement à partir de l'exécution, sans opérations préalables.

Ces divers égarements ont coûté beaucoup de temps dans la réalisation du projet, et ce temps n'a donc pas pu être consacré aux développements ultérieurs que nous désirions entamer.

5. Résultat obtenu.

5.1. L'affichage des informations obtenues.



Ecran de parcourt d'une classe « Hello » héritant de Jframe.

5.2. Son évolution possible.

Comme nous l'avons dit précédemment (4.4.2.), ce projet souffre d'un manque de temps de développement, ce qui se répercute sur les fonctionnalités présentes et l'affichage graphique.

5.2.1. Evolution fonctionnelle.

Le projet obtenu peut évidemment être complété par des améliorations fonctionnelles, notamment au niveau de la pertinence des informations, par la création d'une classe Visiteur pour la récupération des informations « protected ».

Des fonctions de gestion graphiques sont également à prendre en compte pour rendre son ergonomie plus attractive.

5.2.2. Evolution graphique.

Un écran de paramétrage du mode d'affichage de l'arbre et des informations s'y trouvant serait à rajouter.
Ainsi qu'un écran sur la partie droite récupérant des informations plus détaillées sur la classe ou informations sélectionnée dans l'arbre de gauche.

6. Conclusion.

Nous avons, à notre avis, touché du doigt grâce à ce projet à beaucoup de points sensibles sur la réalisation d'un projet.

Il nous a fallu tout d'abord tenter de maîtriser un outil inconnu pour pouvoir répondre de manière la plus efficace aux attentes du projet, nous avons qu'en si peu de temps ce n'était pas facile.

Puis nous avons été confronté à un problème qui à notre avis est majeur dans la bonne marche d'un projet de n'importe quel type : la différence d'interprétation qui peut être faite entre ce que désir le commanditaire et ce que perçoivent les développeurs. Nous nous sommes rendus compte que les choses méritent d'être parfois bien plus explicités, même si elles paraissent claires pour tous, dans l'optique d'un développement efficace.

Ce manque de prise d'information aura nuit à notre projet, hélas. Mais il nous aura apporté beaucoup dans la prise de conscience de la rigueur nécessaire à tout développement, à toutes les étapes.

Nous espérons tout de même avoir atteint notre objectif, au moins d'un point de vue pédagogique.